Data Dependencies

Not all loops can be parallelized. Before adding OpenMP directives need to check for any dependencies:

We categorize three types of dependencies:

- Flow dependence: Read after Write (RAW)
- Anti dependence: Write after Read (WAR)
- Output dependence (Write after Write (WAW)

FLOW	**ANTI**	**OUTPUT**
X = 21	*PRINT *, X*	*X = 21*
*PRINT *, X*	*X = 21*	*X = 21*

Data Dependencies (2)

For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

Let's find the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```

Data Dependencies (2)

For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```

1: S3 → S2 anti (B)

Data Dependencies (2)

For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```

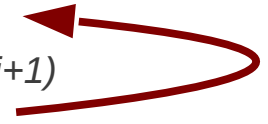
1: S3 → S2 anti (B)
2: S3 → S4 flow (A)

Data Dependencies (2)

For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```



1: S3 → S2 anti (B)
2: S3 → S4 flow (A)
3: S4 → S2 flow (temp)

Data Dependencies (2)

For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```

1: S3 → S2 anti (B)
2: S3 → S4 flow (A)
3: S4 → S2 flow (temp)
4: S4 → S4 output (temp)

Data Dependencies (2)

For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```

1: S3 → S2 anti (B)
2: S3 → S4 flow (A)
3: S4 → S2 flow (temp)
4: S4 → S4 output (temp)

Sometimes it helps to ”unroll” part of the loop to see loop carried dependencies more clear

```
S2: B(1) = temp
S3: A(2) = B(2)
S4: temp = A(1)

S2: B(2) = temp
S3: A(3) = B(3)
S4: temp = A(2)
```

Data Dependencies (2)

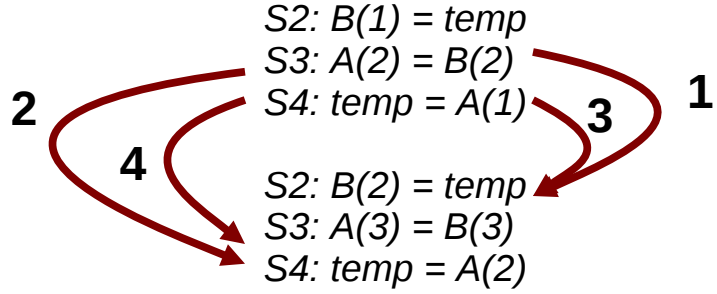
For our purpose (openMP parallel loops) we only care about loop carried dependencies (dependencies between instructions in different iterations of the loop)

What are the dependencies in the following loop?

```
S1: DO I=1,10
S2:    B(i) = temp
S3:    A(i+1) = B(i+1)
S4:    temp = A(i)
S5: ENDDO
```

1: S3 → S2 anti (B)
2: S3 → S4 flow (A)
3: S4 → S2 flow (temp)
4: S4 → S4 output (temp)

Sometimes it helps to ”unroll” part of the loop to see loop carried dependencies more clear



Case Study: Jacobi

Implement a parallel version of the Jacobi algorithm using OpenMP. A sequential version is provided.

Data Dependencies (3)

Loop carried anti- and output dependencies are not true dependencies (re-use of the same name) and in many cases can be resolved relatively easily.

Flow dependencies are true dependencies (there is a flow from definition to its use) and in many cases cannot be removed easily. Might require rewriting the algorithm (if possible)

Resolving Anti/Output Deps

Use PRIVATE clause:

Already saw this in example hello_threads

Rename variables (if possible):

Example: in-place left shift

```
DO i=1,n-1                DO i=1,n-1                !$OMP PARALLEL DO
  A(i)=A(i+1)     →         ANEW(i) = A(i+1)    →   DO i=1,n-1
ENDDO                     ENDDO                       ANEW(i) = A(i+1)
                                                    ENDDO
                                                    !$OMP END PARALLEL DO
```

If has to be in-place can do it in two steps:

```
!$OMP PARALLEL
!$OMP DO
   T(i) = A(i+1)
!$OMP END DO
!$OMP DO
    A(i) = T(i)
!$OMP END DO
!$OMP END PARALLEL
```

More about shared/private vars

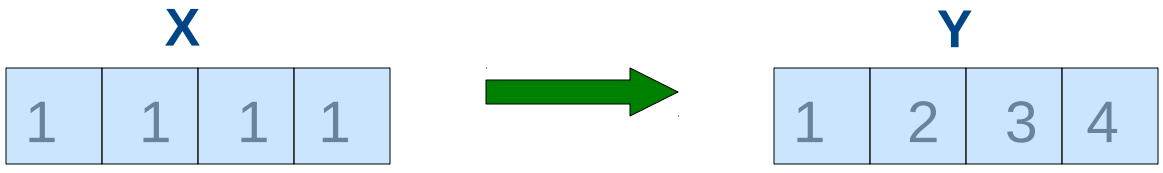
Besides the clauses described before OpenMP provides some additional datascope clauses that are very useful:

- FIRSTPRIVATE (***list***):
  Same as PRIVATE but every private copy of variable 'x' will be initialized with the original value (before the omp region started) of 'x'
- LASTPRIVATE (***list***):
  Same as PRIVATE but the private copies of the variables in list from the last work sharing will be copied to shared version. To be used with !$OMP DO Directive.
- DEFAULT (SHARED | PRIVATE | FIRSTPRIVATE | LASTPRIVATE):
  Specifies the default scope for all variables in omp region.

NOTE: example data scope

Case Study: Removing Flow Deps

Y = prefix(X) → Y(1) = X(1); Y(i) = Y(i-1) + X(i)

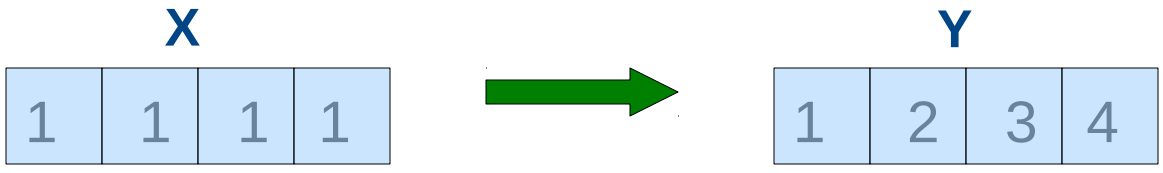


SEQUENTIAL

```
Y[1] = X[1]
DO i=2,n,1
   Y[i] = Y[i-1] + X[i]
ENDDO
```

Case Study: Removing Flow Deps

Y = prefix(X) → Y(1) = X(1); Y(i) = Y(i-1) + X(i)



SEQUENTIAL

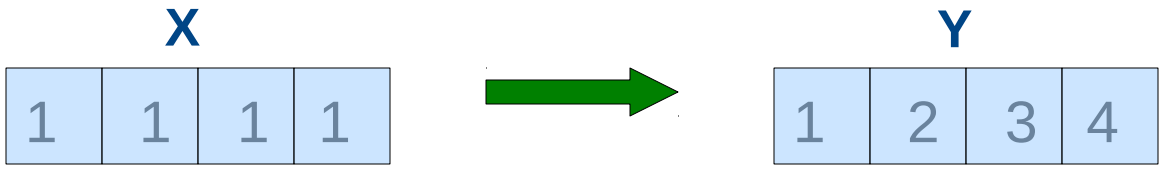
```
Y[1] = X[1]
DO i=2,n,1
   Y[i] = Y[i-1] + X[i]
ENDDO
```

PARALLEL

```
Y[1] = X[1]
!$OMP PARALLEL DO
DO i=2,n,1
   Y[i] = Y[i-1] + X[i]
ENDDO
!$OMP END PARALLEL DO
```

Case Study: Removing Flow Deps

Y = prefix(X) → Y(1) = X(1); Y(i) = Y(i-1) + X(i)



SEQUENTIAL

```
Y[1] = X[1]
DO i=2,n,1
   Y[i] = Y[i-1] + X[i]
ENDDO
```

PARALLEL

```
Y[1] = X[1]
!$OMP PARALLEL DO
DO i=2,n,1
   Y[i] = Y[i-1] + X[i]
ENDDO
!$OMP END PARALLEL DO
```

WHY?

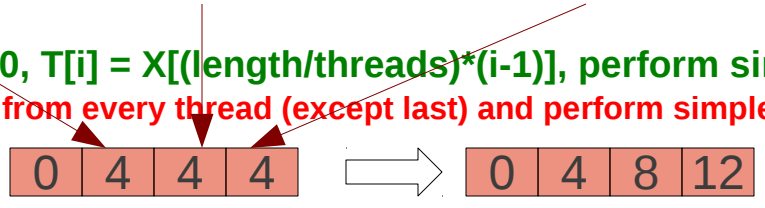
Case Study: Removing Flow Deps

REWRITE ALGORITHM

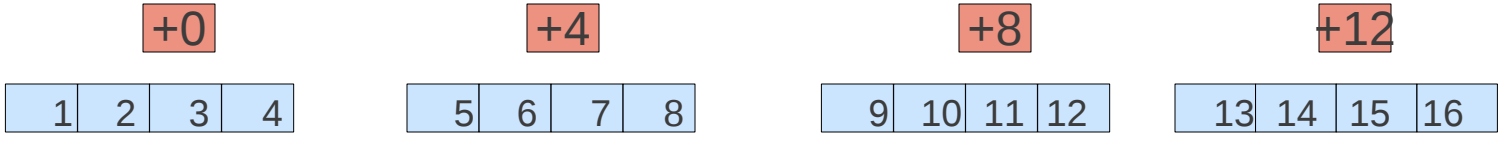
STEP 1: split X among threads; every thread computes its own (partial) prefix sum

STEP 2: create array T → T[1]=0, T[i] = X[(length/threads)*(i-1)], perform simple prefix sum on T

(will collects last element from every thread (except last) and perform simple prefix sum)



STEP 3: every thread adds T[theadid] to all its element



STEP 4: Finished; we rewrote prefex sum by removing dependencies.

Prefix Sum Implementation

How to implement the algorithm on the previous slide?

- Three separate steps
- Steps 1 and 3 can be done in parallel
- Step 2 has to be done sequential
- Step 1 has to be performed before step 2
- Step 2 has to be performed before step 3

NOTE: For illustration purposes we can assume array length is multiple of #threads

NOTE: exercise prefix

Case Study: Removing Flow Deps

This Case study showed an example of an algorithm with real (flow) dependencies

- Sometimes we can rewrite algorightm to run parallel
- Most of the time this is not trivial
- Speedup much less impressive (often)